GLOVE 기능 확장을 위한 자유수면 렌더링 플러그인 설계

2018. 7

황 규 현

한국과학기술정보연구원

목 차

1. 개요

최근 계산 환경의 고성능화로 인해 다양한 분야에서 산출되는 데이터의 양이 매우 방대해지고 있으며, 이를 몰입형 가상현실 환경에서 가시화하기 위한 도구들이 널리 활용되고 있다. 특히 ANSYS 사의 EnSight[1], Kitware 사의 ParaView[2]와 같은 상용/오픈소스 도구들의 경우 자사의 가시화 도구를 몰입형 가상현실 환경에서 활용할 수 있는 기능을 지원하고 있다. 하지만 이러한 도구들의 경우 대용량 데이터의 가시화 결과를 몰입도 있게 렌더링하거나, 방대한 크기의 가시화 결과들을 빠른 응답속도로 조작하는데 제한적이다.

이 문서는 방대한 크기의 가시화 결과를 몰입형 가상현실 환경에서 고품질/고해상도로 가시화하고, 이를 빠른 응답속도로 제어하기 위한 렌더링 플러그인과 프로토콜의 설계에 대해 설명한다. 렌더링 플러그인은 대용량 데이터 가시화 시스템인 GLOVE (GLOVE: GLObal Virtual reality Environment for scientific visualization)[3]의 가상현실용 확장 기능인 GLOVE VR에서 고품질 렌더링을 위해 사용되며, 설계된 프로토콜을 이용하여 고해상도로 렌더링된 결과를 빠른 응답속도로 전송할 수 있도록 하였다.

가. GLOVE VR 소개

고성능 컴퓨팅 환경을 위한 과학적 가시화 시스템인 GLOVE는 가시화 연산을 수행하는 서버와 가시화 결과를 출력하는 클라이언트로 구성되어 있다. GLOVE 서버는 스토리지에 저장된 해석 데이터를 메모리로부터 읽은 후 클라이언트로부터 전달받은 가시화 명령에 따라 해석데이터의 가시화 연산을 수행한다. 가시화 연산을 통해 지오메트리 형태로 추출된 결과는 네트워크를 통해 클라이언트로 전달된다.

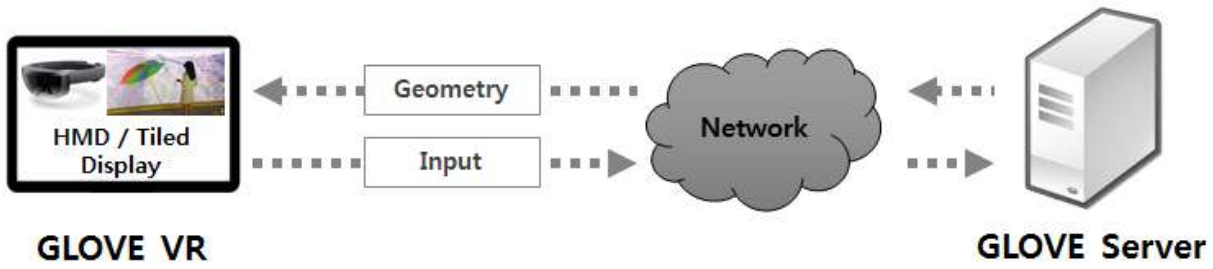

그림 1: GLOVE VR의 가시화 작업 흐름도

그림 1은 GLOVE의 가상현실용 확장 기능인 GLOVE VR의 가시화 작업 흐름도를 나타낸 것이다. 서버-클라이언트 구조인 GLOVE와 동일하게 가시화 연산을 통해 추출된 지오메트

리 데이터를 네트워크를 통해 전달 받은 후, 가상현실 환경에서 사용되는 미들웨어인 UDK를 이용하여 렌더링하고 그 결과를 HMD (Head Mounted Display)나 타일 디스플레이(tiled display) 에 출력하도록 설계되어 있다. 하지만 이러한 구조의 도구들은 방대한 크기의 지오메트리 데이터를 송수신하기 위해 많은 네트워크 전송시간이 요구되며, 렌더링 과정이 수행될 클라이언트의 성능에 따라 빠른 응답속도로 렌더링 결과를 제어하기 어려운 문제가 있다.

나. GLOVE VR 기능 확장을 위한 렌더링 플러그인과 프로토콜

CFD 등 다양한 분야에서 산출되는 대용량 데이터를 가시화 하는 경우 연산 과정에서 수백만 개 이상의 크기를 갖는 폴리곤들이 생성되기 때문에 서버-클라이언트 구조의 가시화 도구에서는 이를 고품질로 렌더링하고 빠른 응답 속도로 렌더링하는데 제약이 있다. 이러한 문제를 해결하기 위해 그림 2와 같이 GLOVE 서버 중 일부를 가시화 전용 서버로 사용하여, 고품질 렌더링과 가시화 결과의 압축 전송 및 관리하는 기능을 수행하도록 설계하였다. 또한 고품질 렌더링을 수행하기 위해 렌더링용 미들웨어인 UDK를 이용하여 렌더링 결과의 품질이 향상될 수 있도록 설계하였으며, 가시화 결과의 압축 전송을 위한 프로토콜을 설계하여 방대한 크기의 가시화 결과를 빠른 응답 속도로 제어할 수 있도록 설계하였다.

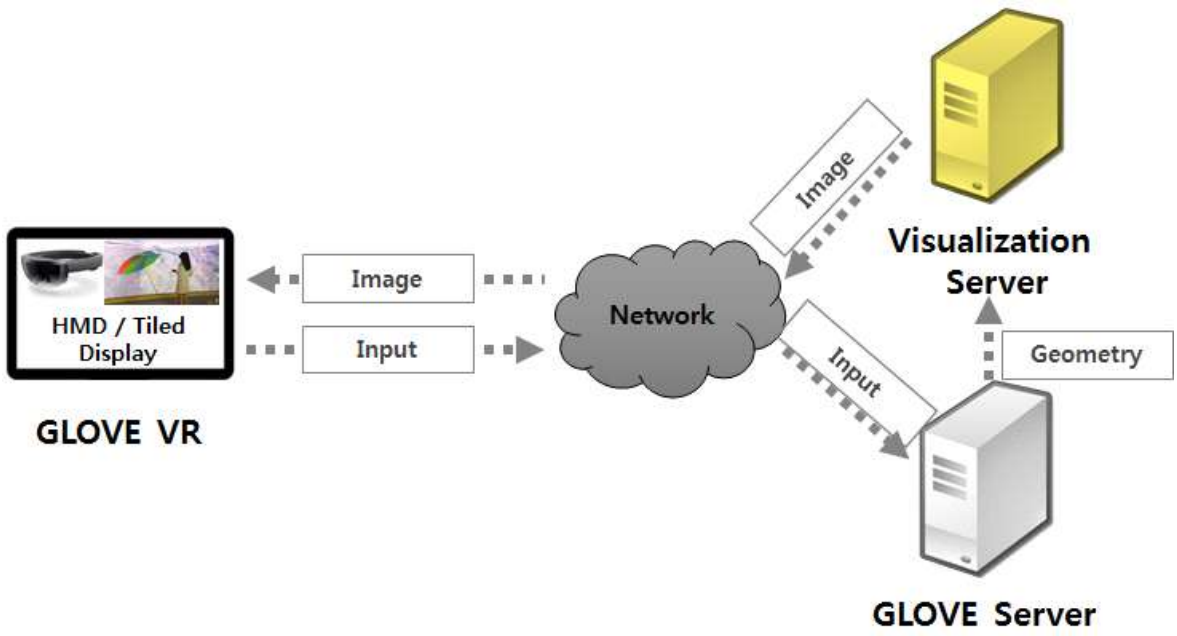

그림 2: 렌더링 플러그인을 적용한 GLOVE VR의 가시화 작업 흐름도

2. GLOVE VR 기능 확장을 위한 렌더링 플러그인 설계

대용량 데이터의 가시화 결과를 실시간에 렌더링하고 빠른 응답 속도로 조작 가능하도록 가시화 연산 결과의 바이너리 형태 변환과 로딩, 그리고 렌더링 기능들을 추가하였다. 추가된 기능들은 렌더링 미들웨어인 UDK의 플러그인 형태로 탑재되어 사용할 수 있도록 설계하였다.

가. 가시화 데이터 변환 및 PMC 생성

① 가시화 데이터 파싱

폴리곤 형태를 갖는 가시화 결과 데이터를 GLOVE 가시화 서버에서 렌더링하기 적합한 형태로 데이터를 가공하기 위해 그림 3과 같은 함수를 이용하여 데이터의 정점과 filler 값, 그리고 법선 등을 추출하게 된다.

```
void ObjectFileLoader::ReadVertexInfo(
   char *buff, std::vector<float>& vPos,
   std::vector<float>& vColor, std::vector<float>& vNorm,
   std::vector<float>& vTex)
{
   float tmpX, tmpY, tmpZ, tmpR, tmpG, tmpB;
   char spChar = buff[1];
   std::vector<std::string> tmpBuff;

   switch (spChar) {
      case ' ': // parse vertex position & colors (v)
         sscanf_s(&buff[2], "%f %f %f %f %f %f", &tmpX, &tmpY, &tmpZ,
                                                 &tmpR, &tmpG, &tmpB);
         vPos.push_back(tmpX); vPos.push_back(tmpY); vPos.push_back(tmpZ);
         if (mIsColorObjModel) {
            vColor.push_back(tmpR); vColor.push_back(tmpG);
            vColor.push_back(tmpB); vColor.push_back(1.0f);
         }
         break;

      case 'n': // parse vertex normals (vn)
         sscanf_s(&buff[2], "%f %f %f", &tmpX, &tmpY, &tmpZ);
         vNorm.push_back(tmpX); vNorm.push_back(tmpY); vNorm.push_back(tmpZ);
         break;

      case 't': // parse texture coordinates (vt)
         sscanf_s(&buff[2], "%f %f", &tmpX, &tmpY);
```

```
            vTex.push_back(tmpX); vTex.push_back(tmpY);
            break;

        default:
            break;
    }
}
```

그림 3: 정점과 법선 등 속성 파싱 함수의 예

추출된 정점과 법선 등의 정보를 이용하여 렌더링될 화면에 출력된 가시화 객체의 면을 구성하기 위해 그림 4와 같은 과정이 수행된다. 면을 구성할 각 정점들의 인덱스 정보를 이용하여 오른손 좌표계 (right handed coordinate)로 구성된 가시화 결과들을 가시화 서버에서 사용하는 왼손 좌표계 (left handed coordinate)로 변환하는 과정이 추가로 수행된다.

```
void ObjectFileLoader::ReadFaceInfo(
    std::string buff,  std::vector<uint32> &vIdx,
    std::vector<uint32> &nIdx, std::vector<uint32> &tIdx)
{
    std::vector<std::string> fType, fInfo;
    this->split(this->tail(buff), fType, " ");
    std::vector<uint32> tmpVertIdx, tmpNormIdx, tmpTexIdx;

    for (int i = 0; i < fType.size(); i++) {
        this->split(fType[i], fInfo, "/");

        int tmpType = 2; // v/vt case
        int fInfoSize = (int)fInfo.size();
        if (fInfoSize > 2) {
            tmpType = 3; // v//vn case
            if (fInfo[1] != "") tmpType = 4;
        }

        switch (tmpType) {
            case 1:    // v1
                tmpVertIdx.push_back(std::stoi(fInfo[0]) - 1);
                break;
            case 2:    // v1/t1
                tmpVertIdx.push_back(std::stoi(fInfo[0]) - 1);
                tmpTexIdx.push_back(std::stoi(fInfo[1]) - 1);
                break;
            case 3:    // v1//n1
                tmpVertIdx.push_back(std::stoi(fInfo[0]) - 1);
                tmpNormIdx.push_back(std::stoi(fInfo[2]) - 1);
```

```
            break;
        case 4:    // v1/t1/n1
            tmpVertIdx.push_back(std::stoi(fInfo[0]) - 1);
            tmpTexIdx.push_back(std::stoi(fInfo[1]) - 1);
            tmpNormIdx.push_back(std::stoi(fInfo[2]) - 1);
            break;
        default:
            break;
      }
   }

   this->IndexSwap(tmpVertIdx, vIdx);
   this->IndexSwap(tmpTexIdx, tIdx);
   this->IndexSwap(tmpNormIdx, nIdx);
}
```

그림 4: 면 속성 파싱 및 좌표계 변환 함수의 예

② 바이너리 저장을 위한 인덱스 생성 및 저장

파싱 과정을 통해 가시화 서버의 공유 메모리에 저장된 가시화 데이터들을 미들웨어에서 빠르게 접근하고 렌더링하기 위해 데이터들을 바이너리 형태로 변환하는 과정이 수행된다. 바이너리 형태로 변환하는 과정에서 불필요한 정점과 법선 벡터들을 삭제되며 실제 가시화 객체의 면을 구성하는 정보만 바이너리 형태로 가시화 서버의 공유 메모리에 저장된다. 그림 5는 바이너리 저장을 위한 인덱스 생성의 예를 나타낸 것으로 생성된 인덱스와 정점, 법선 벡터, 그리고 filler의 값들이 그림 6과 같은 방법을 이용하여 공유 메모리에 저장되도록 하였다.

```
void ObjectFileLoader::GenMergeIndices(
   std::vector<float> &vPos, std::vector<float> &vColor,
   std::vector<float> &vNorm,
   std::vector<float> &vTex, std::vector<uint32> &vIdx,
   std::vector<uint32> &nIdx, std::vector<uint32> &tIdx)
{
   // get norm & uv status
   bool useNorm = !vNorm.empty();
   bool useUV = !vTex.empty();
   bool useVertColor = !vColor.empty();

   if (!useNorm && !useUV) {
      this->CopyMeshInfoToTArray(vPos, vNorm, vColor, vIdx);
      return;
   }
```

```
    // assumes that vIdx = norm idx = uv idx
    for (unsigned int i = 0; i < (unsigned int)vIdx.size(); i++) {
        uint32 tmpIdx = vIdx[i];
        mVertices.Emplace(FVector(vPos[tmpIdx * 3 + 0],
                          vPos[tmpIdx * 3 + 1], vPos[tmpIdx * 3 + 2]));

        if (useVertColor) {
            mVerticesColor.Emplace(FLinearColor(
            vColor[tmpIdx * 4 + 0], vColor[tmpIdx * 4 + 1],
            vColor[tmpIdx * 4 + 2], vColor[tmpIdx * 4 + 3]));
        }

        if (useNorm) {
            tmpIdx = nIdx[i];
            mNormals.Emplace(FVector(vNorm[tmpIdx * 3 + 0],
                             vNorm[tmpIdx * 3 + 1], vNorm[tmpIdx * 3 + 2]));
        }

        if (useUV) {
            tmpIdx = tIdx[i];
            mUVs.Emplace(FVector2D(vTex[tmpIdx * 2 + 0],
                                   vTex[tmpIdx * 2 + 1]));
        }
        mFaces.Emplace(i);
    }
}
```

그림 5: 바이너리 저장을 위한 인덱스 생성 함수의 예

```
void ObjectFileLoader::WriteMeshToFile(std::string filePath)
{
    FILE *fp = NULL;
    errno_t err = fopen_s(&fp, filePath.c_str(), "wb");
    if (err != 0) return;

    int32 vNum = mVertices.Num();
    fwrite(&vNum, sizeof(int32), 1, fp);
    fwrite(mVertices.GetData(), sizeof(float), mVertices.Num() * 3, fp);

    int32 cNum = mVerticesColor.Num();
    fwrite(&cNum, sizeof(int32), 1, fp);
    fwrite(mVerticesColor.GetData(), sizeof(float), mVerticesColor.Num() * 4, fp);

    int32 nNum = mNormals.Num();
    fwrite(&nNum, sizeof(int32), 1, fp);
```

```
    fwrite(mNormals.GetData(), sizeof(float), mNormals.Num() * 3, fp);

    int32 tNum = mUVs.Num();
    fwrite(&tNum, sizeof(int32), 1, fp);
    fwrite(mUVs.GetData(), sizeof(float), mUVs.Num() * 2, fp);

    int32 fNum = mFaces.Num();
    fwrite(&fNum, sizeof(int32), 1, fp);
    fwrite(mFaces.GetData(), sizeof(int32), mFaces.Num(), fp);

    fclose(fp);
}
```

그림 6: 바이너리 파일 저장 함수의 예

③ PMC 생성 및 로딩

그림 7은 가시화 서버의 미들웨어에서 렌더링을 위해 사용되는 PMC (procedural mesh component)를 생성하는 함수를 나타낸 것이다. PMC 함수 생성에 사용되는 가시화 데이터는 바이너리 형태로 공유 메모리에 저장되어 있으며, 바이너리 가시화 데이터 로드를 위해 그림 8과 같은 함수가 사용된다.

```
void AActorPmcBase::CreatePmc()
{
    std::string path(TCHAR_TO_UTF8(*MeshFilePath));

    ObjectFileLoader *objLoader =
      new ObjectFileLoader(path, IsColorModel, IsMergedIdxModel);
    if (objLoader->IsValidFile()) {
       meshComponent->CreateMeshSection_LinearColor(0,
            objLoader->GetVerts(), objLoader->GetFaces(),
            objLoader->GetVertsNormals(), objLoader->GetVertsUVs(),
            objLoader->GetVertsColor(), TArray<FProcMeshTangent>(), false);
    }
    delete objLoader;

    RootComponent->SetRelativeLocation(RelativeLocation);
    RootComponent->SetRelativeScale3D(RelativeScale);
}
```

그림 7: PMC 생성 함수의 예

```
void PMCLoader::ReadProceduralMesh(FILE *fp)
{
   int32 elementNum = 0;

   fread(&elementNum, sizeof(int32), 1, fp);
   if (elementNum > 0) {
      mVertices.AddUninitialized(elementNum);
      this->ReadAndSetDataf(fp, elementNum * 3, mVertices.GetData());
   }

   fread(&elementNum, sizeof(int32), 1, fp);
   if (elementNum > 0) {
      mVerticesColor.AddUninitialized(elementNum);
      this->ReadAndSetDataf(fp, elementNum * 4, mVerticesColor.GetData());
      mIsColorObjModel = true;
   }

   fread(&elementNum, sizeof(int32), 1, fp);
   if (elementNum > 0) {
      mNormals.AddUninitialized(elementNum);
      this->ReadAndSetDataf(fp, elementNum * 3, mNormals.GetData());
   }

   fread(&elementNum, sizeof(int32), 1, fp);
   if (elementNum > 0) {
      mUVs.AddUninitialized(elementNum);
      this->ReadAndSetDataf(fp, elementNum * 2, mUVs.GetData());
   }

   fread(&elementNum, sizeof(int32), 1, fp);
   if (elementNum > 0) {
      mFaces.AddUninitialized(elementNum);
      this->ReadAndSetData(fp, elementNum, mFaces.GetData());
   }
}
```

그림 8: Procedural mesh 데이터 로딩 함수의 예

나. 고품질 자유수면 가시화를 위한 렌더링 플러그인

자유수면의 렌더링 품질 향상을 위해 자유수면 특성을 고려한 렌더링 방법이 수행되어야 한다. 특히 자유수면의 고품질 렌더링 결과를 얻기 위해서는 수면의 움직임에 따라 수면 아래에서 발생되는 빛의 왜곡에 대한 표현이 중요하다. 그림 9와 그림 10은 자유수면의 아래에서 발생되는 빛의 왜곡을 고려한 렌더링 스크립트를 나타낸 것이다.

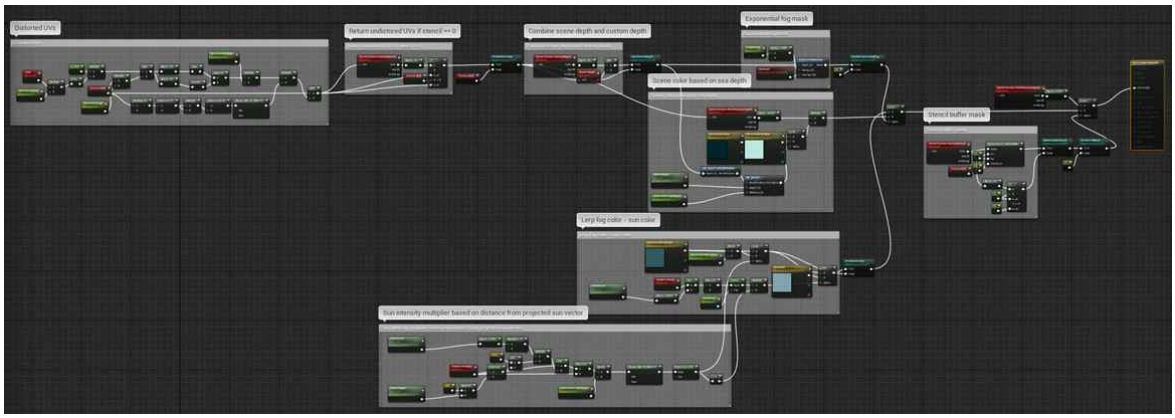

그림 9: 수중 렌더링을 위한 스크립트

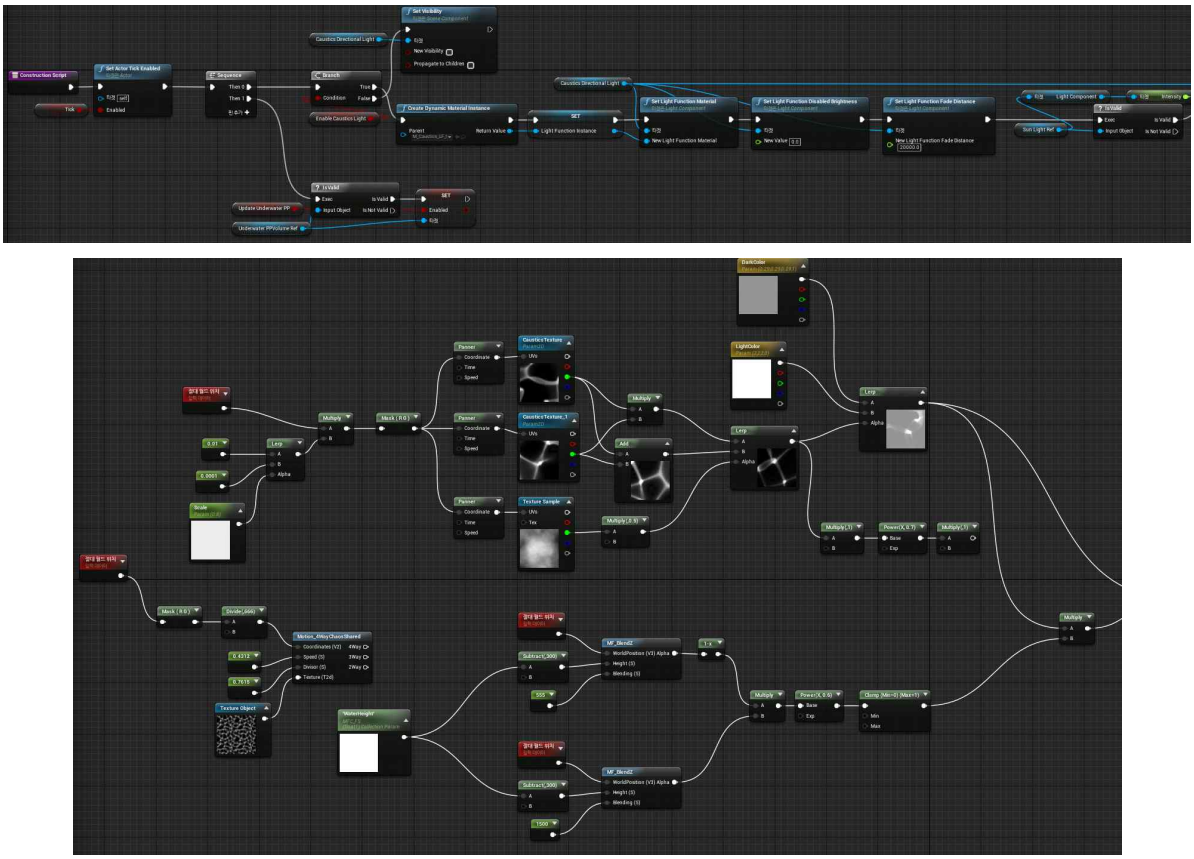

그림 10: 수중 Caustics 표현을 위한 스크립트

그림 11은 앞에서 설명한 자유수면의 표면과 수중 렌더링을 위한 기능들을 적용한 결과를 나타낸 것으로, 빛의 굴절에 따라 수중에서 발생되는 수중 왜곡과 caustics 등의 효과를 실시간에 표현할 수 있도록 설계 하였다.

그림 11: 자유수면 표면 및 수중 렌더링 결과

그림 11은 실세계를 촬영한 HDR(High Dynamic Range) 영상을 조명을 활용하여 가시화 객체의 음영을 사실적으로 표현하기 위한 스크립트를 나타낸 것이다. HDR 영상은 실세계 조명들에 대한 빛의 정보를 높은 정확도로 표현할 수 있기 때문에 이를 이용한 영상기반 조명 (image-based lighting)을 통해 실제와 유사한 자연스러운 음영 표현이 가능하다.

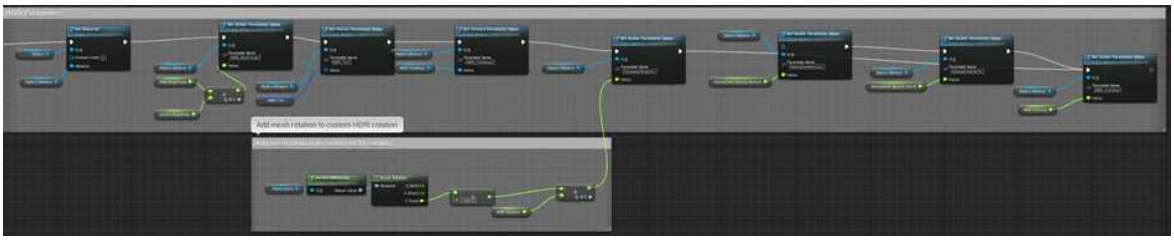

그림 12: HDR 영상을 조명으로 활용하기 위한 스크립트

그림 12는 자유수면과 배 형상, 그리고 프로펠러 뒤쪽의 후류 유동을 가시화 결과에 HDR 영상을 이용한 영상기반 조명을 적용한 결과이다. 영상기반 조명을 적용하지 않은 결과에 비해 자유수면의 표면과 후방 유동 등에서 음영이 정교히 표현되는 것을 확인할 수 있다.

그림 13: HDR을 이용한 영상기반 조명 결과 (좌: 미적용 결과, 우: 적용 결과)

3. 렌더링 결과의 압축 전송을 위한 프로토콜 설계

CFD 등 다양한 분야에서 산출되는 대용량 데이터는 가시화 연산을 통해 수십에서 수백만 개 이상의 폴리곤들을 생성하게 되며, 이를 고해상도로 렌더링한 결과 역시 수백에서 수천만 개의 픽셀들로 구성된다. 가상현실 환경에서는 몰입도 향상을 위해 이러한 가시화 결과들을 빠른 응답속도로 제어하고 출력하는 것이 중요하다. 하지만 서버-클라이언트 구조를 갖는 기존의 도구들의 경우 방대한 크기의 가시화 결과들을 압축 없이 네트워크를 통해 전송하기 때문에 가시화 결과들을 빠른 응답속도로 제어하고 출력하는데 한계가 있다.

몰입형 가상현실 환경을 지원하는 GLOVE VR 역시 서버-클라이언트 구조이기 때문에 기존의 도구들과 동일한 한계를 갖는다. 특히 대용량 데이터의 고품질/고해상도 가시화 결과를 네트워크 전송과 지연시간 등의 이유로 실시간에 렌더링하고 조작하기 어려운 문제가 있다. 이러한 문제를 개선하여 고품질/고해상도 가시화 결과를 빠른 응답속도로 조작하고 출력하기 위해서는 가시화 결과의 실시간 압축 전송 및 저지연 캡춰 기술이 필수적이다.

본 장에서는 고품질/고해상도 가시화 스트림 결과를 실시간에 인코딩/디코딩하고, 빠른 응답속도로 가시화 결과를 전송하기 위한 프로토콜에 대해 설명한다.

가. 압축 전송을 위한 송수신 프로토콜 정의

그림 15는 본 장에서 설명할 가시화 스트림 압축 전송을 위한 프로토콜의 흐름도를 나타낸 것이다. 클라이언트에서 요청된 가시화 작업은 네트워크를 통해 가시화 서버로 전달되며, 가시화 서버는 HPC 노드들에게 해당 가시화 작업을 요청하게 된다. 가시화 작업의 수행 결과인 메시 형태의 데이터는 가시화 서버의 공유 메모리에 저장되며, 가시화 서버에서는 이를 실시간 렌더링한 후 H. 264 형식의 압축 영상으로 인코딩하여 클라이언트로 전송하는 구조이다.

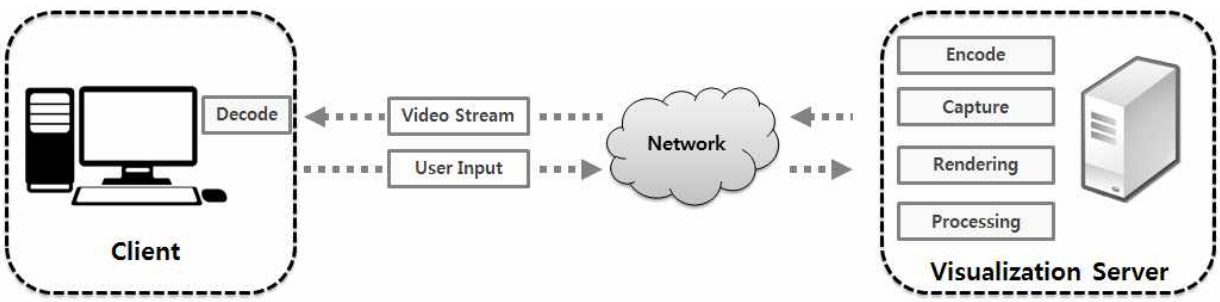

그림 14. 가시화 스트림 압축 전송을 위한 프로토콜 흐름도

1) 프로토콜 구조

가시화 스트림 압축 전송에 사용되는 송수신 프로토콜의 구조는 그림 16과 같다. 프로토콜은 헤더 부분과 실제 송수신될 정보가 기록될 데이터 부분으로 구분된다. 헤더 부분은 메시지 송수신 과정에 필요한 식별자(identifier)와 명령어, 그리고 전송될 실제 데이터의 크기 정보가 기록된다. 데이터 부분은 프로토콜을 통해 전달되는 실제 데이터가 기록되는 부분으로 헤더에 기록된 명령어의 종류에 따라 고정 크기를 갖는 가시화 객체의 모델-뷰 변환 정보나, 가변 크기의 압축된 가시화 결과 등이 기록된다.

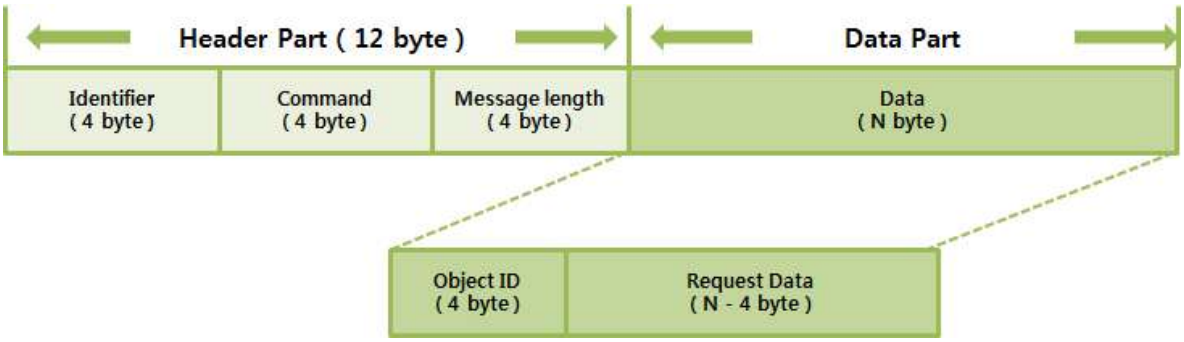

그림 15. 송수신 프로토콜의 구조

프로토콜의 헤더에 저장되는 각 정보들의 크기와 형식은 표 1과 같다. 총 12 바이트 크기를 갖는 프로토콜의 헤더는 4 바이트 크기의 char 형식인 식별자와 4 바이트 크기의 unsigned int 형식인 명령어, 그리고 4 바이트 크기의 unsigned int 형식인 데이터 길이로 구성된다.

표 1. 프로토콜의 헤더 구조

Name	Type	Byte	Contents
Identifier	char	1	A
	char	1	V
	char	1	A
	char	1	!
Command	unsigned int	4	OP_CODES
Data Length	unsigned int	4	Data Length

2) 프로토콜 명령어

표 2는 프로토콜에 사용되는 명령어의 목록 중 일부를 나타낸 것으로 서버-클라이언트 구조를 갖는 GLOVE VR의 특성에 맞게 접속과 종료, 그리고 동기화 및 가시화 결과의 제어 등을 위한 명령어 들이 포함되어 있다.

표 2: 프로토콜의 명령어 목록

Function	ID	Function	ID
Connect	0x0100	Disconnect	0x0200
Sync View	0x1100	Sync Request	0x1200
Sync Data	0x1300		
Streaming Data	0x2100	Object Toggle	0x2200
Translate	0x3100	Rotate	0x3200
Scale	0x3300	Zoom	0x3400

① Connect / Disconnect

GLOVE VR의 확장 기능을 수행할 클라이언트가 가시화 서버에 접속하기 위해 Connect 명령어를 사용한다. Connect와 Disconnect 명령어는 별도의 데이터 필드가 존재하지 않으며, 해당 명령어에 따른 응답 메시지는 존재하지 않는다.

표 3: 접속과 종료 명령어

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length

② Sync Request / Sync View

복수의 클라이언트에서 가시화된 결과 화면을 공유하거나 제어하기 위해 Sync Request 명령어와 Sync View 명령어가 사용된다. 클라이언트 간의 연동 요청은 표 4와 같이 Sync Request 명령어가 사용되며, 이에 대한 응답 메시지는 표 5와 같은 Sync View 메시지를 수신하게 된다. 즉, Sync Request를 이용하여 요청된 동기화가 정상적으로 수행되었는지

여부를 Sync View 메시지를 통해 확인할 수 있다. 동기화가 정상적으로 수행된 경우 가시화 객체의 제어를 위해 Sync Data 명령을 통해 동기화된 각 화면들의 렌더링 결과를 제어할 수 있다.

표 4: Sync Request 요청 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	char array	4	Status (On/Off)

표 5: Sync View 상태 응답 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	char array	4	Status (sync mode on/off)

③ Sync Data

동기화된 화면간의 제어를 위해 Sync Data 메시지가 사용된다. 동기화된 화면들에 출력되는 가시화 결과의 제어를 위해 데이터 부분에 4×4 모델-뷰 변환 행렬과 현재 선택된 가시화 객체의 ID가 전송된다. 선택된 가시화 객체의 ID를 이용하여 해당 객체의 3차원 공간에서의 변환 (회전, 이동, 크기 변환 등)을 수행하게 된다. Sync Data 요청에 따른 응답은 Streaming Data 응답 메시지를 통해 전송 받게 되며 압축된 가시화 영상을 실시간에 디코딩 후 화면에 출력하도록 설계하였다.

표 6: Sync Data 요청 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	unsigned int	4	Object ID
	float array	64	4×4 Model-view Matrix

④ Streaming Data

Sync Data 요청 메시지의 응답 메시지인 Streaming Data 메시지는 HPC에서 가시화 연산 수행을 통해 얻은 메시 데이터를 가시화 서버에서 렌더링한 결과를 송신하기 위한 메시지이다. 가시화 서버에서 렌더링된 결과는 H. 264 형식으로 인코딩 후 Streaming Data 명령어를 이용하여 클라이언트로 전송하게 되는데, 이때 압축된 스트리밍 데이터의 크기가 가변적이기 때문에 표 7과 같은 구조를 이용한다.

표 7: Streaming Data 응답 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	unsigned char	N	Streaming Data

⑤ Object Toggle

화면에 출력되는 가시화 객체를 선택하고 제어하기 위해 현재 선택된 가시화 객체의 ID를 가시화 서버에 알려주는 것이 필요하며, 이를 위해 Object Toggle 명령어가 사용된다. Object Toggle 메시지는 데이터 부분에 선택된 객체의 ID와 활성화 상태가 기록되며, 요청에 따른 Object Toggle 상태를 요청과 동일한 응답 메시지로 전송하게 된다.

표 8: Object Toggle 요청 및 응답 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	unsigned int	4	Object ID
	unsigned char	1	Toggle Status

⑥ Translate

선택된 가시화 객체의 3차원 공간에서의 이동 변환을 위핸 Object Translate 메시지가 사용되며, 메시지의 데이터 부분에는 선택된 대상 객체의 ID와 총 12 바이트의 실수형인 각 축으로의 이동 정보가 기록된다. Object Translate 요청 메시지에 대한 응답은 Streaming Data 메시지를 통해 압축된 스트리밍 결과로 얻게 되며, 이를 위해 가시화 서버는 전달 받은 이동 정보를 식 1을 통해 이동 행렬로 만든 후 가시화 객체에 적용하게 된다.

$$M_t = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 0 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(식 1)}$$

표 9: Object Translate 요청 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	unsigned int	4	Object ID
	float	4	Translate X
	float	4	Translate Y
	float	4	Translate Z

⑦ Rotate

선택된 가시화 객체의 3차원 공간에서의 회전 변환을 위해 Object Rotate 메시지가 사용된다. Object Rotate 메시지의 데이터 부분에는 선택된 대상 객체의 ID와 총 16 바이트 크기의 실수형인 사원수 형태로 표현되는 회전 정보가 기록 된다. 사원수 형태로 표현된 회전 정보는 식 2를 통해 회전 행렬로 변환 되며, 가시화 서버에서는 변환된 회전 행렬을 이용하여 가시화 객체를 회전한 후 Streaming Data 응답 메시지를 이용하여 압축된 가시화 결과를 수신한다.

$$M_q = \begin{bmatrix} 1-2(Q_2^2+Q_3^2) & 2(Q_1Q_2-Q_0Q_3) & 2(Q_1Q_3+Q_0Q_2) & 0 \\ 2(Q_1Q_2+Q_0Q_3) & 1-2(Q_1^2+Q_3^2) & 2(Q_2Q_3-Q_0Q_1) & 0 \\ 2(Q_1Q_3-Q_0Q_2) & 2(Q_2Q_3+Q_0Q_1) & 1-2(Q_1^2+Q_2^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 2)}$$

표 10: Object Rotate 요청 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	unsigned int	4	Object ID
	float	4	W (Q_0)
	float	4	X (Q_1)
	float	4	Y (Q_2)
	float	4	Z (Q_2)

⑧ Scale

선택된 가시화 객체의 3차원 공간에서의 크기 변환을 위해 Object Scale 메시지가 사용된다. Object Scale 메시지의 데이터 부분에는 선택된 대상 객체의 ID와 총 12 바이트 크기의 실수형인 각 축으로의 크기 변환 정보인 S_x, S_y, S_z 가 기록된다. 가시화 서버에서는 식 3을 이용하여 크기 변환 행렬을 생성하게 되며, 크기 변환이 적용된 가시화 객체는 Streaming Data 응답 메시지를 이용하여 압축된 가시화 결과를 수신한다.

$$M_s = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 3)}$$

표 11: Object scale 요청 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	unsigned int	4	Object ID
	float	4	S_x
	float	4	S_x
	float	4	S_x

⑨ Zoom in/out

동기화된 화면의 확대와 축소를 위해 Zoom 명령어가 사용되며 표 12와 같은 요청 메시지를 통해 전달 받은 값을 이용하여 가시화 서버에서 확대와 축소가 수행된다. 확대와 축소가 적용된 가시화 결과는 Streaming Data 응답 메시지를 이용하여 압축된 형태로 수신된다.

표 12: Zoom in/out 요청 메시지

Name	Type	Byte	Contents
Header	char array	4	Identifier
	unsigned int	4	Command
	unsigned int	4	Message Length
Data Part	float	4	Zoom

나. HD급 가시화 스트리밍 결과의 인코딩/디코딩

대용량 데이터의 가시화 연산 과정을 통해 생성된 방대한 크기의 폴리곤 데이터를 빠른 응답 속도로 제어하고 압축 결과를 실시간에 전송하기 위해 FBO (Frame Buffer Object)를 이용한 가시화 스트리밍 결과의 인코딩/디코딩 과정이 수행된다. FBO는 렌더링된 최종 결과가 기록되는 버퍼들의 집합을 의미하는 것으로 하나의 FBO는 여러 개의 렌더링 버퍼(색상과 깊이 버퍼 등)가 포함된다. 기본적으로 렌더링 버퍼는 텍스춰 형태로 사용 및 관리되며, 렌더링 버퍼의 집합인 FBO를 사용하여 화면에 출력된 렌더링 결과를 빠른 응답 속도로 캡춰하고 후처리 과정에 활용하는 것이 가능하다.

1) FBO를 이용한 저지연 프레임버퍼 캡춰

가시화 객체의 렌더링 결과들을 빠른 속도로 캡춰하고 활용하기 위해 색상과 깊이 정보가 기록될 버퍼들을 하나의 FBO로 설정한 후 이를 렌더링 과정에 사용한다. 또한 가시화 결과를 프레임 버퍼의 텍스춰에 직접 렌더링하는 RTT (Render-To-Texture)를 이용하여 색상과 깊이 버퍼들을 효과적으로 관리할 수 있도록 하였다.

```
bool avaRenderFBO::initFBO()
{
   glGenTextures(2, mTexID);
   glBindTexture(GL_TEXTURE_RECTANGLE, mTexID[0]);
   glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR);
   glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR);
   glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, mRenderWidth,
                mRenderHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);

   glBindTexture(GL_TEXTURE_RECTANGLE, mTexID[1]);
   glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR);
   glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR);
   glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_DEPTH_COMPONENT,
                mRenderWidth, mRenderHeight, 0, GL_DEPTH_COMPONENT,
                GL_UNSIGNED_BYTE, NULL);

   glGenFramebuffers(1, &mFBO);
   glBindFramebuffer(GL_FRAMEBUFFER, mFBO);
   glFramebufferTexture(GL_DRAW_FRAMEBUFFER,
                        GL_COLOR_ATTACHMENT0, mTexID[0], 0);
   glFramebufferTexture(GL_DRAW_FRAMEBUFFER,
```

```
                        GL_DEPTH_ATTACHMENT, mTexID[1], 0);

   // frame buffer status check
   if (glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER) !=
            GL_FRAMEBUFFER_COMPLETE) {
      return false;
   }

   glBindFramebuffer(GL_FRAMEBUFFER, 0);
   return true;
}
```

그림 16: Framebuffer object의 초기화 및 설정 과정의 예

그림 17은 FBO의 초기화 및 설정과 관련된 소스코드 일부를 나타낸 것으로 색상과 깊이 버퍼를 드로잉 버퍼에 연결한 후, 그림 18과 같은 방법을 통해 렌더링하게 된다. 최종적으로 프레임 버퍼에 렌더링된 결과는 그림 19에서와 같은 방법으로 압축을 위한 인코딩 함수로 전달된다.

```
void avaRenderFBO::drawFBO()
{
        glBindFramebuffer(GL_FRAMEBUFFER, mFBO);
        glUseProgram(mShaderProgram);

        glClearColor(0.0, 0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
        gluLookAt(0.0, 0.0, 30.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

        glPushMatrix();

        GLfloat m[4][4];
        mTrackball.getMatrix(m);
        glMultMatrixf(&m[0][0]);

        glCallList(mMeshObj[2 * mDispMeshID + 0]);
        glCallList(mMeshObj[2 * mDispMeshID + 1]);

        glPopMatrix();

        glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

그림 17: FBO 렌더링을 위한 함수의 예

```
void avaIFR::render()
{
   mRenderFBO->drawFBO();
   while (mTransferCnt - mReceivedTransfer > mOutStandingTransfer) {
        mDataReleasedEvent.wait();
   }

   mNvIFR.nvIFROGLTransferFramebufferToH264Enc(mTransferObjHandle,
       NULL, mRenderFBO->getFBO(), GL_COLOR_ATTACHMENT0, GL_NONE);

   mTransferStartEvent.signal();
   mTransferCnt++;

   glutSwapBuffers();
}
```

그림 18: FBO 캡춰를 위한 함수의 예

2) 하드웨어 가속 기능을 이용한 실시간 인코딩

네트워크를 기반으로 송수신되는 가시화 스트리밍 영상을 텍스트나 이미지 데이터에 비해 방대한 양의 정보를 포함하고 있기 때문에 스트리밍 영상 전송을 위한 실시간 인코딩 과정이 필수적이다. H. 264 형식의 인코딩 방식으로 스트리밍 영상을 압축할 경우 MPEG4 ASP (Advanced Simple Profile)에 비해 약 40% 가량 압축 성능이 향상되는 것으로 알려져 있다. 하지만 H. 264/AVC 표준은 다른 비디오 표준과 마찬가지로 가변길이 부호화 방식을 따르기 때문에 스트리밍 영상을 인코딩할 때 발생하는 패킷 데이터는 영상의 특징에 따라 큰 폭으로 변할 수 있다. 따라서 고정된 대역폭을 갖는 네트워크를 통해 가시화 스트리밍 영상을 전송할 때 가변적인 패킷 데이터에 대한 고려가 중요하다.

```
bool avaIFR::initIFR()
{
   if (!mNvIFR.initialize()) return false;
   if (mNvIFR.nvIFROGLCreateSession(&mSessionHandle, NULL) !=
      NV_IFROGL_SUCCESS) return false;

   NV_IFROGL_H264_ENC_CONFIG conf;
   memset(&conf, 0, sizeof (NV_IFROGL_H264_ENC_CONFIG));

   conf.profile = 100;
   conf.frameRateNum = 30;
   conf.frameRateDen = 1;
```

```
    conf.width = mFrameWindth;
    conf.height = mFrameHeight;
    conf.avgBitRate = calculateBitrate(mFrameWindth, mFrameHeight);
    conf.GOPLength = 100;
    conf.rateControl = NV_IFROGL_H264_RATE_CONTROL_CBR;
    conf.stereoFormat = NV_IFROGL_H264_STEREO_NONE;
    conf.preset = NV_IFROGL_H264_PRESET_LOW_LATENCY_HP;

    if (mNvIFR.nvIFROGLCreateTransferToH264EncObject(mSessionHandle, &conf,
        &mTransferObjHandle) != NV_IFROGL_SUCCESS) return false;

    return true;
}
```

그림 19: 인코딩을 위한 초기화의 예

그림 20은 가변적 패킷 데이터를 고려한 FBO 렌더링 영상의 인코딩을 위한 초기화 과정을 나타낸 것으로 H. 264로 인코딩된 가시화 스트리밍 영상의 경우 렌더링 결과의 압축 전송을 위한 설계된 프로토콜의 Streaming Data 메시지를 이용하여 그림 21과 같은 함수를 이용하여 송신하게 된다.

```
unsigned int
CommSendThread(void *userData)
{
    int tmpDataSize = 1280 * 720 * 4; // for H264
    char *tmpData = new char[tmpDataSize];
    memset(tmpData, 0, tmpDataSize);
    int recvSize;

    avaIFR::IFR_Data *threadData = (avaIFR::IFR_Data *)userData;
    uintptr_t dataSize;
    const void *data;

    while (!gNetwork->waitForDataSock());

    while (!threadData->IFR_TerminateThread) {
        Sleep(1);

        while (gIFR->mTransferCnt == gIFR->mReceivedTransfer) {
            threadData->IFR_TransferStartedEvent->wait();
            if (threadData->IFR_TerminateThread) break;
        }

        if (gIFR->mNvIFR.nvIFROGLLockTransferData(
```

```
            threadData->IFR_TransferObjHandle, &dataSize, &data)
            != NV_IFROGL_SUCCESS) {
            return 1;
        }

        if (gIsSyncMaster) {
            gNetwork->sendStreamData((char*)data, dataSize);
            gNetwork->sendSyncData((char *)data, dataSize);

        } else {
            gNetwork->recvSyncData(tmpData, &recvSize);

            if (gNetwork->isSyncMode() == 0) {
                gNetwork->sendStreamData((char*)data, dataSize);
            } else {
                gNetwork->sendStreamData(tmpData, recvSize);
            }
        }

        if (gIFR->mNvIFR.nvIFROGLReleaseTransferData(
            threadData->IFR_TransferObjHandle) != NV_IFROGL_SUCCESS) {
            return 1;
        }

        gIFR->mReceivedTransfer++;
        threadData->IFR_DataReleasedEvent->signal();
    }
    return 0;
}
```

그림 20: 인코딩 결과의 송신을 위한 예

3) 하드웨어 가속 기능을 이용한 실시간 디코딩

가시화 서버로부터 프로토콜의 Streaming Data 메시지를 이용하여 전달된 가시화 스트리밍 결과는 그림 22와 같은 방법을 이용하여 디코딩 된다. 디코딩 과정을 통해 무부호형 char 형태로 수신된 인코딩 데이터를 화면에 출력될 영상 프레임 형태 변환하게 되며, 가변적인 패킷 데이터로 구성된 인코딩 데이터의 특성을 고려하여 인코딩 데이터 외에 스트림 영상의 데이터 크기가 인자로 사용된다.

```
void avaCudaDecoder::decodeFrame(unsigned char *data, int len)
{
   if (mFrameParserPtr == NULL) return;
   CUVIDSOURCEDATAPACKET pkt;

   if (len == 0) {
      pkt.flags = CUVID_PKT_ENDOFSTREAM;
      pkt.payload_size = 0;
      pkt.payload = NULL;
      pkt.timestamp = 0;
      cuvidParseVideoData(mFrameParserPtr->mParser, &pkt);
      return;
   }

   pkt.flags = 0; pkt.timestamp = 0;
   pkt.payload_size = (unsigned long)len; pkt.payload = data;
   cuvidParseVideoData(mFrameParserPtr->mParser, &pkt);
}
```

그림 21: 디코딩 과정 수행을 위한 예

그림 23은 그림 22를 이용하여 디코딩된 영상 프레임을 화면에 출력하기 위해 OpenGL 렌더링에 활용할 수 있는 텍스춰 형태로 변환하는 과정을 나타낸 것이다. 텍스춰는 기본으로 HD 해상도에 맞는 크기로 설정되어 있으며 H. 264로 압축된 데이터들을 OpenGL 텍스춰가 처리할 수 있는 ARGB 형태로 변환하는 과정이 포함된다.

```
bool
avaCudaDecoder::cpDecodedFrameToTexture(
      unsigned int &nRepeats,
      int *isProgressive)
{
   CUVIDPARSERDISPINFO dispInfo;
   bool ret = mFrameQueuePtr->dequeue(&dispInfo);
   if (!ret) return false;

   CUdeviceptr pDecodedFrame[2] = { 0, 0 };
   CUdeviceptr pInteropFrame[2] = { 0, 0 };

   CCtxAutoLock aLock(mCtxLock);
   cuCtxPushCurrent(mContext);

   *isProgressive = dispInfo.progressive_frame;
   mIsProgressive = dispInfo.progressive_frame ? true : false;
```

```
    nRepeats = dispInfo.repeat_first_field;
    for (int activeField = 0; activeField < 1; activeField++) {
        CUVIDPROCPARAMS vidProcParams;
        memset(&vidProcParams, 0, sizeof(CUVIDPROCPARAMS));

        vidProcParams.progressive_frame = dispInfo.progressive_frame;
        vidProcParams.second_field      = activeField;
        vidProcParams.top_field_first   = dispInfo.top_field_first;
        vidProcParams.unpaired_field    = true;

        unsigned int nDecodedPitch = 0, nWidth = 0, nHeight = 0;
        mFrameDecoderPtr->mapFrame(dispInfo.picture_index,
              &pDecodedFrame[activeField], nDecodedPitch, &vidProcParams);

        nWidth  = mFrameDecoderPtr->targetWidth();
        nHeight = mFrameDecoderPtr->targetHeight();

        size_t nTexPitch = 0;
        mImageGL->map(&pInteropFrame[activeField], &nTexPitch, activeField);
        nTexPitch = mWindowWidth * 4;

        postProcessframe(&pDecodedFrame[activeField], nDecodedPitch,
              &pInteropFrame[activeField], nTexPitch,
              CudaModulePtr->getModule(), mKernelNv12toARGB, mKernelSID);

        if (mImageGL) mImageGL->unmap(activeField);

        mFrameDecoderPtr->unmapFrame(pDecodedFrame[activeField]);
        mFrameQueuePtr->releaseFrame(&dispInfo);
    }

    checkCudaErrors(cuCtxPopCurrent(NULL));
    return true;
}
```

그림 22: 디코딩 결과의 텍스춰 변환을 위한 함수 예

그림 24는 가시화 스트림 영상을 디코딩하여 ARGB 형태로 변환하는 과정과 화면에 출력하는 과정을 나타낸 것으로, 화면에 출력할 프레임이 디코딩 되었는지 여부를 판단하여 디코딩이 이루어지지 않은 경우 이전 프레임의 디코딩 결과를 화면에 출력하게 된다.

```
void avaCudaDecoder::displayFrame()
{
   static unsigned int nRepeatFrame = 0;
   int repeatFactor = mRepeatFactor;
   int bIsProgressive = 1;
   bool bFrameDecoded = false;

   if (mFrameQueuePtr == NULL) return;

   if (mIsRunning) {
      bFrameDecoded =
         cpDecodedFrameToTexture(nRepeatFrame, &bIsProgressive);
   }

   if (!bFrameDecoded) return;
   while (repeatFactor-- > 0) {
      if (mImageGL != NULL) mImageGL->render(0);
   }
}
```

그림 23: 디코딩 결과의 화면 출력을 위한 함수 예

다. 압축 전송 결과 테스트

개발된 기술의 성능 측정을 위해 상이한 크기는 갖는 두 개의 데이터 셋을 이용하여 인코딩 및 디코딩 성능 테스트를 수행하였다. 실험 환경과 사용된 데이터에 표 13과 같다. 표 14는 다중 연동된 디바이스들의 디코딩에 소요된 시간과 가시화 서버의 인코딩 성능을 나타낸 것으로 가시화 서버에서 인코딩되는 시간은 Fuselage 데이터의 경우 4.35 ms, UCAV 데이터의 경우 4.82 ms가 소요된 것을 확인할 수 있다. 이는 각 데이터의 폴리곤 수에 따른 것으로, 가시화 서버에서 FBO 저지연 렌더링 및 캡춰를 위해 오프 스크린으로 렌더링하는 과정이 수행되기 때문이다. 하지만 두 데이터 모두 인코딩 성능이 약 200 fps, 디코딩 성능이 약 59 fps 이상으로 실시간 렌더링을 의미하는 30 fps 이상의 인코딩/디코딩 성능을 보이기 때문에 가시화 스트리밍 영상의 실시간 인코딩/디코딩 성능을 충족하는 것을 확인할 수 있다.

표 13: 성능 검증을 위한 실험 환경

	인코딩 서버	클라이언트1	클라이언트2	클라이언트3
CPU	Intel Xeon X5690	Intel Core i7-6700	Intel Core i7-4720	ARM Coretex A15
GPU	Quadro M6000	Geforce GTX 980Ti	Geforce GT 750M	Nvidia Tegra K1
메모리	64 GB	32 GB	8 GB	2 GB
네트워크 환경	100 Mbps 이더넷	100 Mbps 이더넷	100 Mbps 이더넷	100 Mbps 이더넷

데이터 이름	가시화 방법	폴리곤 개수
Fuselage	Iso-surface	412,135
UCAV	Iso-surface	1,585,359

표 14: 인코딩/디코딩 실험 결과

	인코딩 소요 시간(ms)	인코딩 성능 (fps)	디코딩 소요 시간(ms)	디코딩 성능 (fps)
Fuselage	4.35	229.88	16.89	59.20
UCAV	4.82	207.46	16.70	59.88

4. 참고문헌

[1] EnSight VR, https://www.ensight.com/ensight-vr, 2018년 7월

[2] ParaView Immersive, https://www.paraview.org/immersive, 2018년 7월

[3] 김민아, 이중연, 허영주, "GLOVE(GLObal Virtual reality visualization Environment for scientific simulation): VR환경에서의 대용량 데이터 가시화 시스템", *2010 한국컴퓨터종합학술대회 논문집*, vol. 37, no. 2(B), pp. 267-271, 2010년.

감사의 말 (Acknowledgements)

본 기술문서는 2017년 정부(미래창조과학부)의 재원으로 국가과학기술연구회 민간융합기술연구사업(no. CMP-17-03-KISTI) 과제의 지원을 받아 작성된 문서임